

IMPROVEMENTS TO ALGORITHMS FOR THE NEAREST NEIGHBOR PROBLEM WITH CYCLICALLY PERMUTED QUERY POINTS

Eduardo F. de Souza Les R. Foulds Humberto J. Longo Wellington S. Martins

Instituto de Informática – Universidade Federal de Goiás

Alameda Palmeiras, Quadra D, Campus Samambaia, CEP 74001-970, Goiânia, Goiás, Brazil

{eduardosouza, lesfoulds, longo, wellington}@inf.ufg.br

RESUMO

O problema do vizinho mais próximo refere-se à localização, em um espaço métrico, de determinado número de pontos de treinamento mais próximos de um conjunto de pontos de consulta. Neste artigo é considerada uma variação desse problema, no qual a menor distância entre um ponto de consulta e um ponto de treinamento é dada por uma pseudométrica definida sobre todas as permutações cíclicas do ponto de consulta. Alguns algoritmos paralelos em CPU foram relatados anteriormente, mas são de implementação difícil devido à impossibilidade de se usar mais *threads* do que o número de pontos de consulta. Agora são apresentados novos algoritmos em GPU, multi-core e manycore, que exploram paralelismo grosso e fino e permitem o uso de mais *threads* do que o número de pontos de consulta. Um extenso estudo experimental dos algoritmos propostos demonstra que conjuntos de dados bastante grandes podem agora ser processados em tempo computacional razoável.

PALAVRAS CHAVE. V_k NN, transformada rápida de Fourier, GPU, multi-core, manycore.

IC – Inteligência Computacional, BDA – Big Data e Analytic, OA – Outras aplicações em PO

ABSTRACT

The nearest neighbor problem is concerned with finding, in a metric space, a number of training points closest to a set of query points. In this paper it is considered a variation of this problem, in which the smallest distance between a reference point and a training point is given by a pseudometric defined over all cyclic permutations of the query point. Some parallel CPU algorithms have been previously reported, but are difficult to implement due to the impossibility of using more threads than the number of query points. New GPU algorithms, multi-core and manycore, which exploit both coarse and fine-grained parallelism and allow the use of more threads than the number of query points are described. An extensive experimental study with the proposed algorithms demonstrates that large datasets can now be processed in reasonable computational time.

KEYWORDS. Nearest neighbors, problem variant, fast Fourier transform, multi-core, many-core.

IC - Computational Intelligence, BDA - Big Data and Analytics, OA - Other applications in PO

1. Introduction

The k -nearest neighbors (denoted here as kNN) is a well-known non-parametric method for classification and regression. Given a natural number k and sets of training and query points in a metric space with distance measure d , the method consists of finding the k training points that are closest to each query point according to d . Any distance measure appropriate for the application at hand can be used, but Euclidean distance is most commonly used and it is applied henceforth. kNN arises in numerous fields of application, including computer vision, computational geometry, database analysis, statistical classification and image processing [Altman, 2012].

A nontrivial variant of kNN (denoted here as $VkNN$), recently introduced [Foulds et al., 2015], concerns the identification of k nearest training points to each query point y , where distance is measured by a pseudometric defined over all the cyclic permutations of y . $VkNN$ arises in rotation-invariant image processing [Kingsbury, 2006], where the need to deal with rotated query points is due to the fact that objects of interest may be in a number of rotated positions, as in aerial photography images. This leads to searching for objects as training points in reference images that are similar to a given number of rotated query point objects – a common operation in rotation-invariant image processing that has been shown to be equivalent to $VkNN$ [Foulds et al., 2015].

An approximate, serial similarity measure based on dual trees and an oriented complex-valued wavelet transform has found to be useful when carrying out the operation discussed above for multi-dimensional image processing [Nelson and Kingsbury, 2011]. However, given the nature of rotation-invariant image processing, it is desirable to obtain exact solutions to the underlying $VkNN$ problem. Furthermore, the resulting set of the required nearest neighbors is a query point-independent operation. This implies that the search for the nearest neighbors of each given query point can be fruitfully conducted in parallel. As the problem parameters are increased, such parallelism tends to decrease computing time compared to serial methods.

We now introduce some concepts from the computer industry (additional concepts are introduced later, as needed). In the last years, the computer industry put less emphasis on the clock frequency of processors and adopted parallelism as the main way to improve performance. So, rather than continuing to improve the speed of a single core, the number of cores was increased in traditional CPUs (Central Processing Units), thus creating multi-core processors. A similar line has been followed in the so-called manycore GPUs (Graphics Processing Units) processors. A typical multi-core CPU contains few complex cores, whereas a manycore GPU may contain several simple cores (count in the thousands) organized into streaming multiprocessors (SMs) groups. The general perception is that processors are becoming wider (i.e. with more cores) rather than faster and that parallelism is an important way to improve performance. CPU cores are often used to speed up the serial processing of instructions, requiring low latency. On the other hand, GPU cores are usually much smaller, have far less complicated circuitry and are more appropriate for high throughput, being able to deal with massive, multi-task parallelism. Although originally intended for graphics computations, the GPUs have become a powerful tool for general-purpose computing. Its high computational power and improved programmability has often made it an excellent choice for high performance scientific computing and big data analytics.

Foulds et al. [2015] recently reported two exact parallel $VkNN$ algorithms. The first is a brute force approach that evaluate the distances between all cyclic permutations of a given query point and a candidate training point. The second algorithm, through a signal-mixing technique, uses

the Discrete Fourier Transform (DFT) to find the closest permutation in one step, saving the need to compute the distances for all permutations. Both are CPU-based parallel algorithms but only map query points processing to CPUs and do not take advantage of fine-grained parallelism whatsoever.

The main contribution of this work is to report improved versions of the original algorithms in [Foulds et al., 2015], enhancing their efficiency with the exploitation of both coarse- and fine-grained parallelism, algebraic transformations (thereby reducing excessive Euclidean distance calculations) and by leveraging the power of GPU processing. An extensive experimental study of the proposed algorithms provide strong evidence that the improved algorithms appear capable of producing exact solutions to large scale, practical datasets in reasonable time and memory.

The remainder of the text is organized as follows. In Section 2 we define $VkNN$ formally and in Section 3 we briefly recall parallel algorithms for $VkNN$. Improvements to the original $VkNN$ algorithms are presented in Section 4. The implementations of the proposed algorithms are discussed in Section 5. Then, in Section 6, we present the results of the computational experiments we conducted. Finally, some concluding remarks and insights into how the current algorithms might be improved even further are presented in Section 7.

2. The $VkNN$ problem

Following the introduction in the first section, let k be a given integer and let $\mathcal{X}$ and $\mathcal{Y}$ be the sets of n training points and r query points, respectively, both in an $\mathbb{R}^m$ metric space with distance measure d . Consider the set of m cyclic permutations of any query point $y \in \mathcal{Y}$. This set can be represented by a circulant matrix C_y [Golub and Loan, 1996], where each row corresponds to a permutation of y . Let y^q be the q^{th} row of C_y , where $1 \leq q \leq m$. In $VkNN$, the measure for computing the distance between a query point $y \in \mathcal{Y}$ and a training point $x \in \mathcal{X}$, denoted by $d_{y,x}$, is defined to be the distance between x and the closest y^q to it, taken over all q , $1 \leq q \leq m$. Thus,

$$d_{y,x} = \min_{q=1,2,\dots,m} d(y^q, x), \quad (1)$$

where $d(a, b)$ is the L^2 norm (Euclidean distance) between points a and b . Note that $d_{y,x}$ is not a metric, but merely a pseudometric because the distance between two distinct points can be zero.

The task in $VkNN$ is to find the k closest training points in $\mathcal{X}$ to each query point $y \in \mathcal{Y}$ with regard to (1). As is well known, the minimization of Euclidean distance between two vectors corresponds to the maximization of their inner product. Hence, the permutation of y that is closest to a given training point x can be determined by the computation of m inner products via the matrix-vector multiplication $C_y \cdot x$. However, the complexity of calculating these distances increases from $\mathcal{O}(mn)$ for kNN to $\mathcal{O}(m^2n)$ for $VkNN$. The commonly used Fourier transform [Frigo and Johnson, 2005] is helpful in the latter case. Fortunately, it has long been known that a circulant matrix can be factored by the discrete Fourier transform (DFT) matrix [Golub and Loan, 1996]. The $C_y \cdot x$ matrix-vector multiplication can be written as

$$C_y \cdot x = DFT^{-1} \cdot ((DFT \cdot y^T) \odot (DFT \cdot x)), \quad (2)$$

where DFT and DFT^{-1} are the DFT matrix and its inverse, respectively, and $\odot$ denote the element-wise vector multiplication. Foulds et al. [2015] have proved that this factorization reduce the complexity of solving the $VkNN$ and have reported an algorithm based on equation (2) with the complexity of just $\mathcal{O}(mn \log(m))$. This algorithm and an improvement to it are discussed later.

3. The original $VkNN$ algorithms

Brief outlines of the CPU-based parallel exact algorithms for $VkNN$ previously reported in [Foulds et al., 2015] are given in this section.

3.1. $BVkNN$

The first, and most intuitive of the original algorithms is a brute-force approach, whose pseudo code is shown in Algorithm 1. The computation associated with each query point is assigned to one CPU thread. So, there is a relation between query points and CPU threads. In terms of algorithm complexity, the bottleneck of this approach is the Euclidean distance computation that occurs multiple times for each training point. The resulting complexity to finding the closest permutation of a query point is $\mathcal{O}(rnm^2)$, where r is the number of query points, n is the number of candidate points, m is the dimensionality of each point and m^2 denotes the cost of computing the Euclidean distance for a pair of points. The increase in dimensionality or in the number of points prohibits this algorithm from handling large-scale datasets in reasonable execution time.

Algorithm 1: $BVkNN$

Input: $\mathcal{Y}$, a set of r query points. $\mathcal{X}$, a set of n training points. k , the number of nearest training points to be found. m , the dimension of the points.

Output: A set $vknnAns$ containing the k nearest neighbors for each query point.

```

1  $vknnAns \leftarrow$  An empty array of size  $r$ .
2 for  $y$  in  $\mathcal{Y}$  do
3    $heap \leftarrow$  A min-heap of size  $k$ 
4   for  $x$  in  $\mathcal{X}$  do
5      $minDist \leftarrow \infty$ 
6      $maxIdx \leftarrow \infty$ 
7     for  $j$  in  $[0, 1, \dots, m - 1]$  do
8        $y_{cl} \leftarrow \text{GenCircLine}(y, j)$ 
9        $dist \leftarrow \text{EuclideanDistance}(y_{cl}, x)$ 
10      if  $dist < minDist$  then
11         $minDist \leftarrow dist$ 
12         $maxIdx \leftarrow j$ 
13       $fibHeap.push([x, minDist, maxIdx]);$ 
14     $vknnAns[i] \leftarrow heap$ 
15 return  $vknnAns$ 

```

The function *GenCircLine* in step 8 is used to generate a circulant line (a permutation) for the query point in question, i.e. y_i , while the function *EuclideanDistance* (step 9) performs the Euclidean distance calculation between the generated circulant line and the training point x_i . A priority queue of size k is used to store the nearest training points for a query point. Thus, assuming that there are T cores, the algorithm will have time complexity of $\mathcal{O}((rnm^2)/T)$ to compute the n distances and $\mathcal{O}((rn \log(k))/T)$ for keeping the k smallest distances.

3.2. FV k NN

This second algorithm improves the BV k NN algorithm by replacing the original $\mathcal{O}(m^2)$ Euclidean distance calculation for every query point by a DFT signal mixing technique, eliminating the need of this calculation for every permutation of a query point, what reduces the complexity to just $\mathcal{O}(m \log(m))$. The pseudo-code for the FV k NN algorithm is shown in Algorithm 2. Instead of performing a linear search, it uses a signal mixing technique that returns a point in the given space. The index of the element with the highest value of this point represents the closest circulant line (cyclic permutation) of a query point to a given training point. This algorithm is quite competitive with BV k NN and better handles the increase in dimensionality, since it is no longer required to calculate the Euclidean distance for every permutation of the query point. Thus, assuming there are T cores, the algorithm takes $\mathcal{O}((rnm \log(m))/T)$ time for computing the distances and $\mathcal{O}((rn \log(k))/T)$ for keeping the k smallest distances.

Algorithm 2: FV k NN

Input: $\mathcal{Y}$, a set of r query points. $\mathcal{X}$, a set of n training points. k , the number of nearest points. m , the dimension size.

Output: A set $vknnAns$ containing the k nearest neighbors for each query point.

```

1  $vknnAns \leftarrow$  An empty array of size  $r$ 
2 for  $y \in \mathcal{Y}$  do
3    $heap \leftarrow$  A min-heap of size  $k$ 
4    $y_{cc} \leftarrow \text{GenCircCol}(y, 0)$ 
5   for  $x \in \mathcal{X}$  do
6      $z \leftarrow \text{ElemWiseProduct}(\text{DFT}(y_{cc}), \text{DFT}(x))$ 
7      $maxIdx \leftarrow \text{FindMaxIdx}(\text{DFT}^{-1}(z))$ 
8      $pt \leftarrow \text{GenCircLine}(y, maxIdx)$ 
9      $minDist \leftarrow \text{EuclideanDistance}(pt, x)$ 
10     $heap.push([x, minDist, maxIdx])$ 
11   $vknnAns[i] \leftarrow heap$ 
12 return  $vknnAns$ 
```

4. Improvements to the original V k NN algorithms

When analyzing V k NN from an algorithmic perspective, one sees that memory data is independent for each query point. The problem involves a given set of query points and an independent set of training points, all of them read-only to the computation. Consequently, the only write operation is to construct the resulting set of nearest neighbors for every query point, which is a query point-independent operation. Naturally, it is possible to take advantage of these layout characteristics to improve the parallel algorithms mentioned earlier.

4.1. Improvements to BV k NN

The most expensive part of the BV k NN algorithm is the loop used to calculate the distances of every permutation of a given query point to a particular candidate point (steps 7–12). It can be improved by replacing the successive Euclidean distance calculations by a matrix multiplication operation to be performed by one of the many fast existing algorithms for this purpose.

Let y be a query point and a and b be distinct candidate training points. The following steps show the derivation of the equivalence of transforming a Euclidean distance calculation to a matrix multiplication calculation:

$$\sqrt{\sum_{i=1}^n (y_i - a_i)^2} < \sqrt{\sum_{i=1}^n (y_i - b_i)^2}. \quad (3)$$

Squaring both sides and expanding:

$$\sum_{i=1}^n (y_i^2 - 2y_i a_i + a_i^2) < \sum_{i=1}^n (y_i^2 - 2y_i b_i + b_i^2). \quad (4)$$

Removing the y_i^2 term from both sides:

$$\sum_{i=1}^n a_i^2 - 2 \sum_{i=1}^n y_i a_i < \sum_{i=1}^n b_i^2 - 2 \sum_{i=1}^n y_i b_i; \quad (5)$$

we obtain a vector notation:

$$\|a\|^2 - 2y^T a < \|b\|^2 - 2y^T b. \quad (6)$$

We can improve the necessary computations even further by dividing both sides of the last equation by -2 . This enables a and b to be preprocessed, but flips the inequality sign. So, the greater values in this case represent the smaller distances.

$$-\frac{\|a\|^2}{2} + y^T a > -\frac{\|b\|^2}{2} + y^T b. \quad (7)$$

In practical terms, we will use only the left side of this last inequality as a will represent the set of candidate training points and y the set of query points. Then, we will look for the highest values (corresponding to the smallest distances) and find the desired k nearest neighbors. We replace the steps 7–12 in the BV k NN code, which compares the circulant lines of a query point with all candidate points, by a matrix multiplication that produces an equivalent result. This replacement allows the use of efficient parallel matrix multiplication algorithms and reduces the overhead originally introduced by the partitioned computation in the steps 7–12. The algorithm obtained by the above algebraic improvements of the B k NN algorithm is denoted by BV k NN-AO.

4.2. Improvements to FV k NN

The BV k NN and FV k NN algorithms take advantage of coarse-grained parallelism. FV k NN is superior to BV k NN for large-scale practical numerical instances as it takes advantage of the cyclic structure of the data and reduces the complexity of finding the closest query permutation from $\mathcal{O}(m^2)$ to $\mathcal{O}(m \log(m))$. However, given the availability of modern parallel architectures and the fact that query points can be processed independently, the exploitation of fine-grained parallelism can further improve the performance of solution algorithms for V k NN. This encouraged us to develop a GPU-based parallel solution to improve the FV k NN algorithm.

Before detailing the proposed algorithm, we recall some other GPU-related concepts. A GPU program promotes parallelism through a data-parallel single program multiple data function, termed a kernel function. During implementation, the programmer can configure the number of threads to be used. Threads execute parallel data computations of the kernel and are organized in groups (thread blocks) that are further organized into a grid structure. When a kernel is launched, the

blocks within a grid are distributed to idle streaming multiprocessors while the threads are allocated to the simple cores. A parallel program in a GPU is usually divided into blocks of computation that run independently on a group of processors (cores) known as SMs – Streaming Multiprocessors. These blocks, in turn, have to be further divided into smaller tasks (threads) that execute on the processors (cores) known as SCs – Streaming Cores, with each thread being able to communicate with other threads in the same block. Each of these threads has access to a larger global memory as well as to a small (but fast) shared memory and registers. The GPU supports thousands of light-weight concurrent threads and, unlike the CPU threads, the overhead of creation and switching is negligible. To overcome the high latency of the global memory, it is important to have more threads than the number of SCs and to have threads able to access consecutive memory addresses that can be easily coalesced. Another important data movement channel is based on a current interconnection technology called PCI Express, whereby a CPU and a GPU can exchange data between each of their address spaces, albeit at a much slower speed. The GPU programming model requires that part of the application runs on the CPU while the computationally-intensive part is accelerated by the GPU.

Algorithm 3: GV k NN

Input: $\mathcal{Y}$, a set of query points. $\mathcal{X}$, a set of training points. k , the number of nearest points. m , the dimension size.

Output: A set $vknnAns$ containing the k nearest neighbors for each query point.

```

1  $vknnAns \leftarrow$  An empty array of size  $k$ 
2  $circCols \leftarrow \text{GenCircCols}(\mathcal{Y})$ 
3  $circCols_{DFT} \leftarrow \text{DFT}(circCols)$ 
4  $\mathcal{X}_{DFT} \leftarrow \text{DFT}(\mathcal{X})$ 
5 for  $circCol \in circCols_{DFT}$  do
6    $z \leftarrow \text{ElemWiseProduct}(circCol, \mathcal{X}_{DFT})$ 
7    $z_{DFT^{-1}} \leftarrow \text{DFT}^{-1}(z)$ 
8    $idxs \leftarrow \text{MaxIdx}(z_{DFT^{-1}})$ 
9    $closestpermutations \leftarrow \text{GenCircLines}(\mathcal{Y}, idxs)$ 
10   $near \leftarrow \text{EuclideanNorm}(closestpermutations, \mathcal{X})$ 
11   $vknnAns[i] \leftarrow \text{SortAndExtract}(near, k)$ 
12 return  $vknnAns$ 
```

A pseudo-code for the proposed fine-grained parallel algorithm is shown in Algorithm 3. It performs a preprocessing operation (steps 1–4) for calculating the DFTs of both the query and training points, and then (steps 6–11) applies different operations for each query point until the solution is obtained, that is, the k nearest neighbors for each query are found. We now explain how parallelism is exploited in the algorithm. In step 2, the first column of the query permutation matrix is calculated in parallel for every query point. This information is needed in the following step (3), where the DFT is calculated in parallel for each query point. Similarly, the DFT is calculated in parallel for each training point in step 4. Having performed this preprocessing, the algorithm performs a series of parallel operations (steps 6–11) for each query point. These operations (described in Section 2) are performed in parallel, associating the query point being processed with every train-

ing point. First, an element-wise vector multiplication (line 6) is calculated. Then the inverse DFT is applied to this result (step 7). In step 8, the index of the closest permutation is calculated, and subsequently used to generate the circulant line of the query permutation matrix being processed. The Euclidean distance is calculated in step 10. Finally, the k nearest training points are obtained by using a GPU parallel sort.

The complexity analysis of the proposed $GVkNN$ algorithm, assuming t cores are available, is as follows. Each column of a cyclically permuted query point can be obtained in $\mathcal{O}(m)$ time. Since there are r queries, step 2 can be done in $\mathcal{O}((rm)/t)$. The DFT takes $\mathcal{O}(m \log(m))$ for each point, thus step 3 takes $\mathcal{O}((rm \log(m))/t)$ and, similarly, step 4 takes $\mathcal{O}((nm \log(m))/t)$. The operations in the loop (steps 6-11) are performed r times. Since each element-wise multiplication of step 6 takes $\mathcal{O}(m)$ sequentially, its parallel complexity is $\mathcal{O}((rm)/t)$. For step 7, the parallel inverse DFT for all training points can be done in time $\mathcal{O}((nm \log(m))/t)$ and for all query points the time is $\mathcal{O}((rnm \log(m))/t)$. For step 8, the index of the maximum scalar product can be found using a parallel reduction and thus takes $\mathcal{O}((n \log(m))/t)$ for each training point and $\mathcal{O}((rn \log(m))/t)$ for all query points. Step 9 is similar to step 2 but deals with n training points, so it takes $\mathcal{O}((rnm)/t)$. The Euclidean distance in step 10 takes $\mathcal{O}(m)$ for each training point and thus it takes $\mathcal{O}((rnm)/t)$ in total. Finally, the sorting in step 11 can be done in $\mathcal{O}(n \log(n))$ but requires $\mathcal{O}((rn \log(n))/t)$ when applied to all query points in parallel. In this algorithm, a GPU parallel sort is used instead of a priority queue because the data is processed independently on the kernels, reducing its complexity. Overall, the $GVkNN$ algorithm is dominated by the time to sort the calculated distances which requires $\mathcal{O}((rn \log(n))/t)$ time.

5. Implementation of the algorithms

In this section, we discuss the implementations of the two original algorithms and our proposed improved versions. The $BVkNN$ and the $FVkNN$ algorithms are mapped to a multi-core CPU. The $BVkNN$ -AO algorithm is also mapped to a multi-core CPU whereas the $GVkNN$ algorithm targets a manycore GPU.

$BVkNN$: The running codes of both $BVkNN$ and its improved version, denoted by $BVkNN$ -AO, were implemented in C++, targeting CPU cores. The first step of these implementations is to generate the cyclic permutations for every query point, avoiding the process of regenerating them when processing each candidate point. Usually the number n of query points is much larger than the number T of CPU cores. So, the $BVkNN$ algorithm maps $\lceil r/T \rceil$ query points to each thread, where each thread is used to perform a batch processing (queue) of the query points. The intermediate processing (e.g. calculating Euclidean distance) is performed with lightweight CPU threads that compete for the same resources. Each thread then processes its own list of query points, calculating the Euclidean distance between a cyclic permutation of the query and every candidate point using lightweight threads. The calculated distance is inserted into a fixed-size heap. When the processing of candidate points is completed, the resulting heap is the representation of the nearest candidate points to a cyclic permutation.

$FVkNN$: The $FVkNN$ implementation also targets a multi-core CPU, exploiting coarse-grained parallelism, and it divides the query points among CPU threads. The significant difference with the last implementation is the use of the DFT technique to reduce the number of Euclidean distances calculated.

BV k NN-AO: The BV k NN-AO implementation follows the same steps as BV k NN, but differs when calculating distances. Instead of calculating the distance for each query point and each candidate point sequentially, BV k NN-AO utilizes the matrix version of the Euclidean distance. Doing this makes it possible to retrieve the distances in one calculation (the matrix multiplication). Thus, parallelism is exploited in the matrix multiplication step.

GV k NN: The GV k NN algorithm targets CUDA GPUs, exploiting fine-grained parallelism. It is based on nine kernels that implement steps 2–4 and 6–11 of the Algorithm 3. Since the computation related to each point can be done independently, GV k NN takes full advantage of GPU parallelism. In most of the nine kernels, the number of blocks is set equal to the number of query (or training) points, but the number of threads is set at 256, an optimal value found experimentally. However, this information is not reported for third party kernels (Thrust and cuFFT libraries [Bell and Hoberock, 2012, Corporation, 2012]) since it depends on many factors. Each of the nine kernels used in our implementation, indexed by its corresponding step, is described next:

- 1 – Circular column (step 2):** generates the first column of the query point permutation matrix. Since the structure of the circulant matrix is known a priori, this kernel simply computes the permuted values for the first column of the matrix. The kernel is launched with r (the number of queries) blocks so that each block uses its threads to generate the first column of the matrix in parallel for a given query point. The number of threads per block was set to 256.
- 2 – Query point DFT (step 3):** NVIDIA [Corporation, 2012] provides an efficient implementation for the DFT operation (cuFFT) and we make use of that to calculate the required DFTs. We use cuFFT in batch mode so that multiple query points can be processed in a single kernel call. This is much more efficient than simply calling the DFT many times in a loop since some of the intermediate results can be reused.
- 3 – Training point DFT (step 4):** this operation is done in a similar manner, we use the cuFFT internal kernel in batch mode to calculate the DFTs of the training points.
- 4 – Element-wise product (step 6):** the element-wise product operation is implemented by a kernel of n training point blocks, each having m (the number of dimensions) threads.
- 5 – Inverse DFT (step 7):** calculate the inverse DFT using cuFFT internal kernel, operating in batch mode.
- 6 – Maximum index (step 8):** finds the maximum index which corresponds to the minimal distance. It is launched with n training point blocks, each also having m threads. The maximum index is found by using a parallel tree reduction. A parallel tree reduction is one of the most widely used parallel computation pattern in which we combine neighboring elements, using a logical tree, until there is only one element left.
- 7 – Circular row (step 9):** generates the row of the query permutation matrix which corresponds to the index found previously. As in the first kernel, it simply computes the permuted values for the row being considered. It also uses n training point blocks, once again, each with m threads.
- 8 – Euclidean distance (step 10):** to calculate the Euclidean distance between the query point being considered and each training point, this kernel is again launched with n training point blocks, having m threads each.
- 9 – Sorting (step 11):** uses a function from the Thrust library to sort the distances so that the k nearest distances can be extracted.

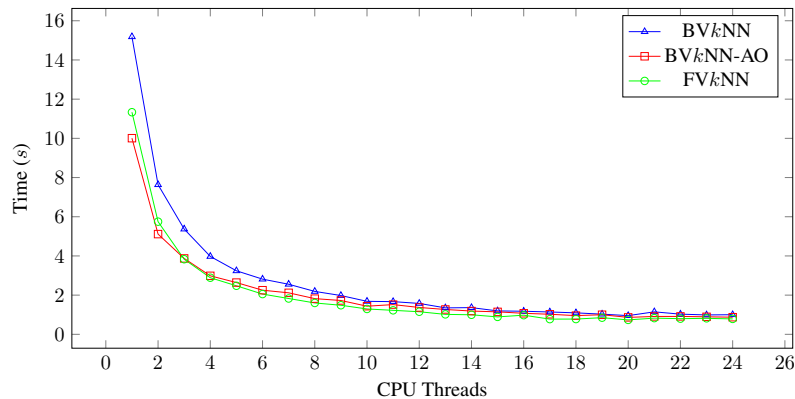
The kernels 1, 2, 4, 6, 7, and 8 exploit two levels of parallelism. Firstly, they associate a block of threads with each query (or training) point and secondly, each of these threads is associated with one of the m elements comprising the point coordinates. If the number m of elements is greater than the number of threads of a block, the elements are split into z parts, where z is the rounded-up ratio between the number of elements and the number of threads in the block. Then, each part is processed separately, in a round-robin fashion. For the calculation of the maximum index, we implemented a parallel binary tree reduction using shared memory. Therefore, we allocated an array in shared memory whose size is equal to the number of threads in the block. The array is divided into two sub-arrays and then the elements of these sub-arrays are compared. The process is repeated until the first position of the array contains the maximum element.

6. Experimental evaluation

The experimental work was conducted on a machine running Ubuntu 16.04 LTS 64-bits, with two Intel Xeon E5-2620 (6 cores per socket, totaling 12 cores and 24 threads), 16 GB of ECC RAM, and one GeForce Zotac Nvidia GTX Titan Black, with 6GB of RAM and 2,880 CUDA cores. The CPU code was compiled with GCC 4.8.5, while the GPU code used the compiler provided by the CUDA Toolkit version 7.5. All the codes targeted the native architecture.

We tested the implementations on randomly generated synthetic datasets as they are shown to be fair representatives of the real-world problems in permutation invariant image processing, as shown by Foulds et al. [2015]. The datasets consists of 100 query points, 10^4 training points, and dimensions ranging from 16 to 4096, increasing as a power of 2. The executions were set to find the $k = 50$ nearest neighbors. The reported results for both BV_kNN , BV_kNN-AO , and FV_kNN are for 12 CPU cores (24 threads). The results for GV_kNN are for one GPU, with 2880 cores. To consider the costs of all data transfers in our experiments, we report the wall times on a dedicated machine so as to rule out external factors, such as high loads caused by other processes. The reported numbers are an average of 10 independent runs.

Figure 1: Exploring the thread number in CPU-based algorithms



We conducted an experiment with an 256-dimensional synthetic dataset to analyze the behavior of the CPU-based algorithms when increasing the number of threads. The results are shown in Figure 1. We notice a significant speedup until around 10 threads. Amdahl's law [Mittal and Vetter, 2015] theorizes about this behavior of a speedup barrier when adding cores (parallelism) to a problem, where beyond a certain point there does not seem to be more improvement. Our second

experiment was designed to compare the performance of the original algorithms and the proposed improvements. Since the algorithms complexity depends on the number of dimensions used, we ran both the CPU-based algorithms, with 24 threads, and the GPU-based algorithm varying the dimensions from 16 up to 4096, to show the scalability of each method. Each column in Table 1 represents the run times obtained (average of running wall-clock time from 10 executions) for a dimension (m) with the listed algorithms.

Table 1: Run time for synthetic datasets results (values in seconds).

Dimension	16	32	64	128	256	512	1024	2048	4096
BVkNN	0.041	0.104	0.296	1.126	3.755	15.704	92.858	417.467	1783.224
BVkNN-AO	0.031	0.082	0.253	0.915	3.346	17.916	76.415	317.263	1294.514
FVkNN	0.173	0.258	0.400	0.733	2.455	5.050	9.013	192.826	225.534
GVkNN	0.649	0.692	0.762	0.928	1.341	2.183	4.468	4.793	5.330

Although having higher algorithmic complexity, the CPU-based algorithms are well suited for low dimensionality datasets. In our experiments, BV k NN-AO outperformed BV k NN with speedups from 1.16 to 1.32. It is the most suitable algorithm for dimensions below 64, with excellent results compared to the other algorithms. There is significant preprocessing required by the DFT approaches which is why FV k NN is slower in low dimensions. However, as BV k NN and BV k NN-AO have quadratic complexity, their execution times quickly increase in dimension size. Indeed, FV k NN outperformed the other algorithms when the dimension was at least 128. DFT-based methods are faster when dimensions are a power of 2 because this size is able to match the high-speed caches in the CPU processors and there is not the necessity to swap much memory between its caches and the main memory (RAM).

Our fine-grained (and most robust) GPU-based algorithm (GV k NN) is the fastest among the mentioned algorithms for dimensions at least 256. Its speedups in relation to FV k NN increase as dimensions grow (ranging from 1.54 up to 42.31 for dimensions above 256), what demonstrate its capability of handling high-dimensional instances reasonably efficiently. The performance profiling showed that most of the execution time of GV k NN ($\approx 80\%$) is spent moving data from RAM to GPU VRAM. The two most time-consuming steps are the dot product operation ($\approx 20\%$ of the time) and the Euclidean distance calculation ($\approx 15\%$ of the time). Given that our kernels associate threads with point coordinates, the more the dimension increase, the more parallelism can be exploited in the GPU.

7. Conclusion

The two improved parallel algorithms presented for solving a particular variant of the k -nearest neighbors problem exploit data parallelism by processing many query points simultaneously. The first algorithm BV k NN-AO, employs algebraic improvements to the CPU-based brute-force approach, exploiting coarse-grained parallelism. It performs well when there are no more than 64 dimensions. The second algorithm GV k NN, improves on the DFT-based algorithm by performing the DFT and the needed distance calculations in a fine-grained GPU-based parallel manner. This exploitation of two levels of parallelism, i.e, inter-query and intra-query, results in a highly threaded, scalable parallel implementation that takes full advantage of modern multi-core (CPU) and manycore (GPU) processors architectures. The algorithms are somewhat complementary and

can be combined to produce a best-of-all-worlds implementation which can switch between them, depending on the size of the input data. BV_kNN -AO is better for small dimensions, while GV_kNN is superior when the number of dimensions exceeds 256. The extensive experimental work conducted to evaluate our implementations both revealed the limitations of the CPU-based algorithm and showed that the GPU-based algorithm is capable of processing quite large data sets in reasonable computation time.

Finally, we suggest some possible lines of further research. A current limitation of our GPU algorithm is the slowness of data transfer between host and device memory, due to the current interconnection technology (PCI Express). Emerging interconnection technologies (e.g. NVLink) promise to alleviate this problem. Another possible way to attempt to overcome the limitation is to construct an algorithm that overlaps distance calculations and data transfer by processing points as soon as they are allocated in the memory. Also, a GPU-based version of the BV_kNN -AO algorithm based on existing efficient GPU-based algebraic algorithms might be a productive way of exploiting fine-grained parallelism. Furthermore, Foulds et al. [2015] proposed a condensation technique that discards candidate training points that are possibly far from the query point permutations. The reduction is often more than 50% of the number of candidate points. It might be productive to construct a GPU-based version of the technique. A more ambitious future work is to develop multi-GPU algorithms that features some, or all, of the above mentioned modifications. Such algorithms might potentially efficiently process datasets of orders of magnitude greater than those currently handled and allow new and improved applications of the V_kNN problem.

References

- Altman, N. S. (2012). An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3).
- Bell, N. and Hoberock, J. (2012). Thrust: Productivity-Oriented Library for CUDA. Astrophysics Source Code Library. <http://adsabs.harvard.edu/abs/2012ascl.soft12014B>.
- Corporation, N. (2012). Cudft library, programming guide.
- Foulds, L. R., de Moraes Neto, J. P., Longo, H. J., do Nascimento, H. A. D., and Martins, W. S. (2015). A variant of k -nearest neighbors search with cyclically permuted query points for rotation-invariant image processing. *Discrete Applied Mathematics*, 197:123–144.
- Frigo, M. and Johnson, S. G. (2005). The design and implementation of FFTW3. *Proc. of the IEEE*, 93(2):216–231.
- Golub, G. and Loan, C. V. (1996). *Matrix Computations*. Johns Hopkins University Press, USA, 3rd edition.
- Kingsbury, N. G. (2006). Permutation-invariant local feature matching with complex wavelets. In *Proc. European Conference on Signal Processing (EUSIPCO)*, p. 901–904.
- Mittal, S. and Vetter, J. S. (2015). A survey of cpu-gpu heterogeneous computing techniques. *ACM Computing Surveys (CSUR)*, 47(4):1–35.
- Nelson, J. D. B. and Kingsbury, N. G. (2011). Enhanced shift and scale tolerance for permutation invariant polar matching with dual-tree wavelets. *IEEE Transactions on Image Processing*, 20: 814–821.